Finding Optimal Formulae for Bilinear Maps

Razvan Barbulescu, Jérémie Detrey, Nicolas Estibals, Paul Zimmermann

A unifying algorithm finding all formulae for bilinear computations

Răzvan Bărbulescu Jérémie Detrey Nicolas Estibals
Paul Zimmermann

10 November 2011
[work in progress]

1 Introduction

Karatsuba's algorithm for polynomial multiplication can be formalized as follows Using linear combinations of the three products $g_0 = a_0b_0, g_1 = a_1b_1, g_2 = (a_0 + a_1)(b_0 + b_1)$, we want to generate the three bilinear forms $c_0 = a_0b_0, c_1 = a_0b_1 + a_1b_0, c_2 = a_1b_1$. One solution is $c_0 = g_0, c_1 = g_2 - g_0 - g_1, c_2 = g_1$.

We can generalize this problem as follows. Consider a field K, and two finite sets of variables $\{a_0, a_1, ..., a_{n-1}\}$ and $\{b_0, b_1, ..., b_{m-1}\}$. For $\kappa \in K$, the multiplications κa_i or κb_j are *scalar products*, while the multiplications a_ib_j are *non-scalar products*. Given a finite input set $\mathcal{G}$ of products of linear forms, each one being of the form $g = (\sum_i \alpha_i a_i)(\sum_j \beta_j b_j)$, and a target set $\mathcal{C} = \{c_0, c_1, \ldots, c_{\ell-1}\}$ of bilinear forms, each element of $\mathcal{C}$ being of the form $\sum \gamma_{i,j} a_i b_j$ with $0 \leq i < n$ and $0 \leq j < m$. The problem consists in generating all the elements of $\mathcal{C}$ by linear combinations of a minimal number k of elements in $\mathcal{G}$, or to find all solutions with that minimal number k.

Example 1. *(Karatsuba [KO62]) We have* $K = \mathbb{Q}$, $n = m = 2$, $\mathcal{G} = \{g_0 := a_0b_0, g_1 := a_1b_1, g_2 := (a_0 + a_1)(b_0 + b_1), g_3 := a_0b_1, g_4 := a_1b_0\}$, *and* $\mathcal{C} = \{a_0b_0, a_0b_1 + a_1b_0, a_1b_1\}$ *which are the bilinear forms involved in the polynomial product* $(a_0 + a_1X)(b_0 + b_1X)$. *In that case the minimal number of non-scalar products is* $k = 3$, *and a solution is given above.*

Example 2. *(Product of* 3*-term polynomials in* GF(2) *[Too63],[CA66]) We have* $K = \mathrm{GF}(2)$, $n = m = 3$, $\mathcal{G} = \{a_0b_0, a_1b_1, a_2b_2, (a_0 + a_1)(b_0 + b_1), (a_0 + a_2)(b_0+b_2), (a_1+a_2)(b_1+b_2), (a_0+a_1+a_2)(b_0+b_1+b_2)\}$ *(we restrict ourselves in this case to symmetric products, i.e., each multiplicand uses the same linear combination of the* a_i *and* b_j*). Here is one of the* 7 *solutions with only* $k = 6$ *out of* 7 *elements from* $\mathcal{G}$. *The needed product are:* $w_0 := b_0a_0, w_1 := b_1a_1, w_2 := b_2a_2, w_3 := (b_0+b_1)(a_0+a_1), w_4 := (b_1+b_2)(a_1+a_2), w_5 := (b_0+$

$b_2)(a_0+a_2)$, *then the coefficients of the polynomial products are reconstructed as follows:* $c_0 := w_0, c_1 := w_0 + w_1 + w_3, c_2 := w_0 + w_1 + w_2 + w_5, c_3 := w_1 + w_2 + w_4, c_4 := w_2$.

Example 3. *(Middle product [HQZ04]) Here* $k = \mathbb{Q}$, $n = 2$, $m = 3$, $\mathcal{G}$ *is the set of all bilinear products of the form* $(\alpha_0 a_0 + \alpha_1 a_1)(\beta_0 b_0 + \beta_1 b_1 + \beta_2 b_2)$, *and* $\mathcal{C} = \{c_0 := a_0 b_1 + a_1 b_0, c_1 := a_0 b_2 + a_1 b_1\}$. *A solution with* $k = 3$ *considers the subset* $\{g_0 := a_0(b_1 + b_2), g_1 := a_1(b_0 + b_1), g_2 := (a_1 - a_0)b_1\}$, *with* $c_0 = g_1 - g_2$ *and* $c_1 = g_0 + g_2$.

Example 4. *(Cross product [BZ10, Ex. 1.18]) The cross-product of two vectors* $[a_0, a_1, a_2]$ *and* $[b_0, b_1, b_2]$ *is* $\mathcal{C} = [c_0 = a_1 b_2 - a_2 b_1, c_1 = a_2 b_0 - a_0 b_2, c_2 = a_0 b_1 - a_1 b_0]$.

If the minus signs are replaced by plus signs in $\mathcal{C}$, *Michel Quercia found a solution with* 5 *products which is valid in any characteristic; another solution with* 4 *products in characteristic odd or zero; and he proved that a solution with* 3 *multiplies cannot exist in characteristic different from* 2.

Example 5. *(Strassen [Str69]) The Strassen matrix multiplication method is nothing more than a recursive use of a trick which multiplies two matrices of size two in* 7 *products. Indeed, let* R *be a ring,* A, $B \in \mathcal{M}_{2^N}(R)$ *for some* N *and put* $C = A \cdot B$. *Split each of* A, B *and* C *in four blocks* $\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in \mathcal{M}_{2^{N-1}}(R)$ *with* $i, j \in [1, 2]$. *A direct verification shows that the four blocks of* C *are obtained as it follows:*

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\
\mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}.
\end{aligned}$$

In order to improve the number of block multiplications, we consider 7 *matrices each of which can be computed by a block product:*

$$\begin{aligned}
\mathbf{G}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{G}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{G}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{G}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{G}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{G}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{G}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}).
\end{aligned} \tag{1}$$

Finally, we obtain the target blocks $C_{i,j}$ *by linear combinations of the* G_k *and thus with no extra product:*

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{G}_1 + \mathbf{G}_4 - \mathbf{G}_5 + \mathbf{G}_7 \\
\mathbf{C}_{1,2} &= \mathbf{G}_3 + \mathbf{G}_5 \\
\mathbf{C}_{2,1} &= \mathbf{G}_2 + \mathbf{G}_4 \\
\mathbf{C}_{2,2} &= \mathbf{G}_1 - \mathbf{G}_2 + \mathbf{G}_3 + \mathbf{G}_6.
\end{aligned} \tag{2}$$

If the same trick is used for each product in (1), then we obtain a method for multiplying two matrices of $\mathcal{M}_{2^N}(R)$ in 7^N multiplications in R.

In the language we use in this paper, the G_k are a subset of the set $\mathcal{G}$ of block products and the $C_{i,j}$ are the target bilinear forms. Here we take $n = 4$ and $m = 4$ since we have 4 blocks in both A and B. The field K needs to contain the scalars in equations (1) and (2) and to have the same characteristic as R. Therefore we take $k = \mathrm{GF}(p)$ if $\mathrm{char}(R) = p$ is prime and $k = \mathbb{Q}$ otherwise.

Example 6. *(Multiplication in $\mathbb{C}$ [Knu81, p. 506]) Here $K = \mathrm{GF}(3)$, $n = 2$ and $m = 2$. We want to multiply two complex numbers $a = a_0 + \mathbf{i}\, a_1$ and $b = b_0 + \mathbf{i}\, b_1$, i.e., the target set is $\mathbb{C} = \{a_0b_0 - a_1b_1, a_0b_1 + a_1b_0\}$. A solution is to take $g_0 = a_0b_0$, $g_1 = a_1b_1$ and $g_2 = (a_0 + a_1)(b_0 + b_1)$ and to obtain $ab = (g_0 - g_1) + \mathbf{i}\,(g_2 - g_0 - g_1)$. Since this trick generalizes to the multiplication of a complex s by a vector of complex numbers $[z_0, \ldots, z_{N-1}]$, a natural question is if one can compute $sz_0, sz_1, \ldots, sz_{N-1}$ in less than $3N$ real×real products.*

More formally for a given integer k, we want to find a subset $\mathcal{W}$ of k elements from $\mathcal{G}$ such that the vector space W spanned by $\mathcal{W}$ contains $\mathcal{C}$. Several solutions $\mathcal{W}$ exist. For Example 2 over $K = \mathbb{Q}$, Montgomery [Mon05] gives a family of solutions:

$$\begin{aligned}(a_0 + a_1X + a_2X^2)(b_0 + b_1X + b_2X^2) &= a_0b_0(C + 1 - X - X^2)\\ +a_1b_1(C - X + X^2 - X^3) &+ a_2b_2(C - X^2 - X^3 + X^4)\\ +(a_0 + a_1)(b_0 + b_1)(-C + X) &+ (a_0 + a_2)(b_0 + b_2)(-C + X^2)\\ +(a_1 + a_2)(b_1 + b_2)(-C + X^3) &+ (a_0 + a_1 + a_2)(b_0 + b_1 + b_2)C,\end{aligned}$$

where C is an arbitrary polynomial in X.

The main contribution of that paper is that, instead of searching for $\mathcal{W}$, we search for the vector space W spanned by $\mathcal{W}$, and then fewer solutions exist. In Example 2, the product $(a_0+a_1+a_2)(b_0+b_1+b_2)$ can be obtained as a linear combination of the other six products, as can be seen by taking $X = 1$ and $C = 0$ in Eq. (3):

$$\begin{aligned}(a_0 + a_1 + a_2)(b_0 + b_1 + b_2) &= (a_0 + a_1)(b_0 + b_1) + (a_0 + a_2)(b_0 + b_2)\\ &+ (a_1 + a_2)(b_1 + b_2) - a_0b_0 - a_1b_1 - a_2b_2.\end{aligned}$$

Thus we are looking for a subspace W of the space V spanned by all products a_ib_j such that: (i) $\mathcal{C} \subset W$; (ii) there exists a subset $\mathcal{W}$ of $\mathcal{G}$ such that W is spanned by $\mathcal{W}$; (iii) the dimension of W is k.

2 A framework to study bilinear computation

- construction of the product space

- reduction of the product space dimension
- express the problem
- reconstruction of formulae
- etc.

3 Enumerating spaces of formulae

In the previous Section we reduced our problem of finding all formulae for some bilinear computation to a linear algebra problem: given a finite set $\mathcal{G}$ of a K-vector space and a target subspace T of it, we are looking for all the subspaces W containing T which can be generated by only elements of $\mathcal{G}$ ($\mathrm{Span}(W \cap \mathcal{G}) = W$). More specifically we are looking for formulae using only k products, this corresponds to limit ourselves to subspaces W of rank k: this is the problem we designed an algorithm for in this Section. One should note that this algorithm is fully independent of the original problem of optimizing bilinear computations and thus is much more general.

3.1 Naive algorithm

The trivial approach of this problem consists in enumerating all the $\binom{\#\mathcal{G}}{k}$ subsets $\mathcal{W} = \{g_1, \ldots, g_k\}$ of $\mathcal{G}$ and test if their span recovers T. The most efficient way to test this inclusion is to construct $W = \mathrm{Span}\{g_1, \ldots, g_k\}$ by its matrix in row echelon form and then test if the echelon form of $T = [t_0, \ldots, t_{\dim T - 1}]$ reduced to 0 in this matrix. This will require $O(\frac{k(k+1)}{2} \dim V)$ operations on the field K for constructing W and the test of each inclusion of t_i will require $O(k \dim V)$ field operations. All in all, the complexity of the naive algorithm is:

$$\mathcal{O}\left(\binom{\#\mathcal{G}}{k} \cdot [\frac{k(k+1)}{2} + k \dim T] \cdot \dim V\right).$$

3.2 Reducing the number of tests

The main drawback of the naive approach is that different sets of products $\mathcal{W}_i$ may be linearly dependent and span the same vector space $W = \mathrm{Span}\, \mathcal{W}_i$. We design an algorithm that takes advantage of this redundancy by looking directly for the vector spaces W (instead of the sets of generators) that recover our goal $\mathcal{T}$. More formally, we search all the subspaces W of V such that:

(i) $T \subset W$ (the target space is recovered);

(ii) $\mathrm{Span}(W \cap \mathcal{G}) = W$ (by spanning only generators in $\mathcal{G}$);

(iii) $\dim W = k$ (and such that only k of those generators are needed.)

A first remark helping to construct our algorithm is that the target space T is contained in the searched spaces. Thus we should look for all the W by extending this vector space. We introduce a relaxed condition to this extend:

$$(ii') \quad \exists \mathcal{I} \subset \mathcal{G} \mid W = T \oplus \mathrm{Span}(\mathcal{I}).$$

Lemma 1. *All subspaces W that verify (i) and (ii) also verify (ii').*

Proof. This lemma is a direct application of Theorem 1. □

Theorem 1 (Generators-Free Family[1])**.** *Let V be a vector space over a field K and $\mathcal{G}$ a finite set of generators. Let $\mathcal{T}$ be a free family of V. Then there exists a subset $\mathcal{H}$ of $\mathcal{G}$ such that $\mathcal{T} \bigcup \mathcal{H}$ is a base of V.*

Proof. If $\mathcal{T}$ generates V then we take $\mathcal{H} = \emptyset$.

Let now assume that $\mathcal{T}$ does not span V. If we had $\mathcal{G} \subset \mathrm{Span}(\mathcal{T})$, then it would follow $V = \mathrm{Span}(\mathcal{G}) \subset \mathrm{Span}(\mathcal{T})$ which contradicts the assumption. Therefore, there exists $g \in \mathcal{G}$ such that $g \notin \mathrm{Span}\, T$. This shows that $\mathcal{T}' := \mathcal{T} \bigcup \{g\}$ is free. We iterate the procedure with $\mathcal{T}'$ replacing $\mathcal{C}$. Since we cannot choose twice the same element $g \in \mathcal{G}$ and since $\mathcal{G}$ is finite, the process terminates. □

It is obvious that if W is a vector space which satisfies the conditions (i) and (ii') then $W \oplus \mathrm{Span}(g)$ also satisfies those conditions for all $g \in \mathcal{G}$. Algorithm 1 explores all the vector spaces constructed by adding some g's to the space T. Our algorithm stops when a sufficient dimension is reached and then the obtained space is tested against the unrestricted condition (ii). Thus all the spaces output verify the three unrestricted conditions.

Let W be a subspace verifying (i), (ii) and (iii). We then choose $\mathcal{H} \subset \mathcal{G}$ be a base of W. By Theorem 1, we may extract a subfamily $\mathcal{I}$ of $\mathcal{H}$ such that $T \oplus \mathrm{Span}(\mathcal{I})$ is W. Consequently Algorithm 1 will find the subspace W. This proves that the algorithm will find all the spaces verifying the three conditions.

Example 7. *Here is an example which shows how Algorithm 1 works. Put $K = \mathbb{Q}$, $V = \mathbb{Q}^3$, $\mathcal{C} = \{c_0, c_1\}$ with $c_0 = (1,1,0)$ and $c_1 = (1,1,1)$ and $\mathcal{G} = \{g_0, g_1, g_2, g_3, g_4\}$ with $g_0 = (1,0,0)$, $g_1 = (0,1,0)$, $g_2 = (0,0,1)$, $g_3 = (0,0,-1)\}$ and $g_4 = (1,0,1)$ (cf. Figure 1).*

The algorithm initializes W to $T = Span(t_0, t_1)$. Since at any moment W is represented by a basis here $w_0 = t_0$, $w_1 = t_1$ and $[w_0, w_1]$ represent W. The algorithm continues by comparing the dimension of W with $k = 2$. It is the case, so it moves on to computing $W \bigcap \mathcal{G} = \{g_2, g_3\}$. Finally, the rank of $[g_2, g_3]$ is tested. Unfortunately, it is only 1, so we cannot make a

[1] Cf. Theorem 3.15 in [Art91].

Algorithm 1 An algorithm to find all the spaces of rank k generated by only elements of $\mathcal{G}$ that contain the target space T.

Input: $T, \mathcal{G}, \dim T \leq k \leq \dim V$
Output: $S = \{W \mid T \subset W \wedge \dim W = k \wedge \mathrm{Span}(W \cap \mathcal{G}) = W)\}$

```
1.  S = ∅
2.  procedure enlarge_to_dim_k(W,H) :
3.      if dim W ≥ k then
4.          if rk(W ∩ G) = k then
5.              S = S ∪ {W}
6.      else
7.          while H ≠ ∅ :
8.              Pop g in H
9.              if g ∉ W
10.                 enlarge_to_dim_k(W ⊕ Span(g), H)
11. enlarge_to_dim_k(T, G \ T)
12. return S
```

basis of W from elements of $\mathcal{G}$ and we discard W. No solution is therefore obtained for $k = 2$

Let now look at case $k = 3$. Here the algorithm starts with $W = T$, but since it has dimension less than 3, W has to be enlarged. The algorithm will need $\mathcal{H} = \mathcal{G} \setminus T$. Here it is $\mathcal{H} = [g_0, g_1, g_4]$. The algorithm adds the first element of $\mathcal{H}$ to W_{old} and obtains $W_{new} = [c_0, c_1, g_0]$. Since dim($W_{new}$) becomes $k = 3$, we compute the set $\mathcal{G} \bigcap W_{new} = [g_0, g_1, g_2, g_3, g_4]$. This set has rank 3, so we have just found a solution. We print W_{new} and come back to W_{old} which can be enlarged with g_1 and g_4.

Finally, consider the case $k = 4$. Here W will be C, then $\mathbb{Q}^3$, which does not contain 4 free vectors of $\mathcal{G}$, so no solution is printed.

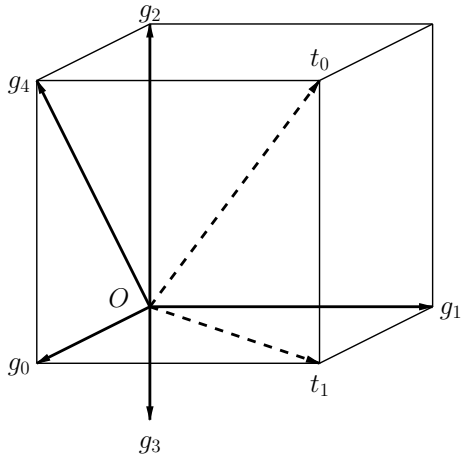
Compare the behaviour of Algorithm 1 to the one of the naive one. For $k = 2$ the naive algorithm considers $\binom{\#\mathcal{G}}{p} = 10$ pairs in $\mathcal{G}$ and for each one tests if the elements of $\mathcal{T}$ are contained. For $k = 3$, the naive algorithm computes $\binom{5}{3} = 10$ triples, while for $k = 4$ it will have 5 tests. In short, the naive algorithm considers subsets of $\mathcal{G}$ and tests if their span contains T, while Algorithm 1 considers sub-spaces above T and tests if they have basis in $\mathcal{G}$.

3.3 Complexity analysis

This last approach allows us to drastically reduce the number of operations needed to find all the formulae to calculate a given bilinear computation because of two facts:

1. searching spaces of formulae instead of sets of formulae;

Figure 1: Example of set $\mathcal{G}$.



2. constructing those spaces from C instead of from $\{0\}$.

In this subsection, we give the complexity analysis of Algorithm 1. The algorithm consists in extending C to a vector space of dimension k by adding $k - \dim C$ vectors in $\mathcal{G}'$. For each of those $\binom{\#\mathcal{G}'}{k-\mathrm{rk}\,\mathcal{C}}$ spaces, we have to construct their echelon form from the echelon form of C, find all the elements of $\mathcal{G}$ it contains, and check its rank. We can thus bound the complexity of the algorithm by:

$$\mathcal{O}\left(\binom{\#\mathcal{G}'}{k-\mathrm{rk}\,\mathcal{C}}\left[\underbrace{\frac{(k+\dim C)(k-\dim C)}{2}}_{\text{construction of } W}+\underbrace{k\#\mathcal{G}}_{\text{computation of } W\cap\mathcal{G}}+\underbrace{\frac{\#G(\#G+1)}{2}}_{\text{verification of the rank}}\right]\cdot\dim V\right).$$

3.4 Optimisations

- echelon form representation
- $\mathcal{H}' = \mathcal{H}/\sim$ where $g \sim g' \equiv W \oplus \mathrm{Span}\, g = W \oplus \mathrm{Span}\, g'$
- DFS/BFS trade-off

4 Lifting the formulae to $\mathbb{Z}$

A close look at algorithm 1 shows that it can be used for all fields as long as the set of generators is finite. Given a bilinear computation (polynomial product, matrix multiplication etc.) with variables $a_0,\ldots,a_{n-1}$, $b_0,\ldots,b_{m-1}$, we can search for formulae over $\mathbb{Q}$ with $\mathcal{G} = \{(\Sigma_{i=0}^{n-1}\mu_i a_i)(\Sigma_{j=0}^{m-1}\nu_j b_j) | \mu \in$

$\{-1,0,1\}^n, \nu \in \{-1,0,1\}^m\}$. If a formula uses no division, then we say it is a formula over $\mathbb{Z}$. Since all the formulae over $\mathbb{Z}$ project into formulae over all finite fields, searching formulae over $\mathbb{Z}$ can be done by searching formulae over $GF(2)$ and then lifting them to $\mathbb{Z}$ by the procedure below.

We keep the notations above and we put $\bar{\mathcal{G}} = \{(\Sigma_{i=0}^{n-1}\mu_i a_i)(\Sigma_{j=0}^{m-1}\nu_j b_j) | \mu \in GF(2)^n, \nu \in GF(2)^m\}$. Next we define $\pi : \mathcal{G} \to \bar{\mathcal{G}}$, which projects modulo 2. Given a formula with the list of products $[h_0, \ldots, h_{k-1}]$ over $GF(2)$ for some k, we put $\mathcal{G}_{lift} = \pi^{-1}(h_0) \bigcup \cdots \bigcup \pi^{-1}(h_{k-1})$. By applying algorithm 1 to $\mathcal{G}_{lift}$ we find all the solutions over $\mathbb{Q}$ and by an easy selection we have all the formulae over $\mathbb{Z}$ which use k products.

If we consider the arithmetics over $\mathbb{Q}$ as elementary operations, then the complexity of the algorithm above is that of algorithm 1 run on $\mathcal{G}_{lift}$. Since for all products h_i over $GF(2)$, $\pi^{-1}(h_i)$ has at most 2^{m+n} elements, the cardinal of $\mathcal{G}_{lift}$ is at most $2^{(m+n)k}$. Or this is the cardinal of $\bar{\mathcal{G}}$, so the time of lifting a formula is dominated by the one of finding all the solutions over $GF(2)$. In the case when limitations are set on the products in $\mathbb{Z}$, i.e. we restricted to $\mathcal{H} \subset \mathcal{G}$, we search for formulae with generators $\pi(\mathcal{H})$ and lift each formula to its preimage in $\mathcal{H}$. Moreover, lifting can be accelerated if algorithm 1 is adapted rather than utilized.

The drawback of the project-solve-lift strategy comes when we accept to make a couple of easy divisions. For example, multiplying two trinomials can be done in four products if we allow division by 2, but an exhaustive search proves that the optimal formula over $GF(2)$ has 6 products. Lifting an optimal formula finds formulae over $\mathbb{Q}$ with 6 products, therefore non-optimal over $\mathbb{Q}$. This brings us to saying that a prime p is lucky for a bilinear computation if all the optimal formulae over $\mathbb{Q}$ project onto optimal formulae over $GF(p)$. For example, 3 is lucky for the trinomials product.

5 Connection with other algorithms of polynomial multiplication

Let K be a field such that $\mathrm{char}(K) \geq 2n-2$ or $\mathrm{char}(K) = 0$. As before, let m and n be integers and $a_0, \ldots, a_{n-1}$, $b_0, \ldots, b_{m-1}$ variables. Put $\mathcal{G} = \{(\Sigma_{i=0}^{n-1}\mu_i a_i)(\sigma_{j=0}^{m-1}\nu_j b_j) | \mu \in K^n, \nu \in K^m\}$ and $\mathcal{T} = \{\Sigma_{i+j=k} a_i b_j | k \in [0..n+m-2]\}$. What is the output of Algorithm 1 on the input $(\mathcal{G}, T, k := 2n-1)$? In order to see this, let us name $x_0, \ldots, x_{2n-1}$ some elements of K and put $x_{2n-1} = \infty$. We notice that $A(x_{2n-1}) \cdot B(x_{2n-1}) = a_{n-1}b_{m-1}$ which belongs to $\mathrm{Span}(\mathcal{T})$. Also, for all $i = 0, 2n-2$, $A(x_i)B(x_i) = (a_0 + a_1 x_i + \ldots + a_{n-1}x_i^{n-1}) \cdot (b_0 + b_1 x_i + \ldots + b_{n-1}x_i^{n-1})$ and therefore belongs to $\mathrm{Span}(\mathcal{T})$. Therefore we have $2n-1$ products in $\mathrm{Span}(\mathcal{T})$. Next we consider $\mathcal{T}$ as a basis of $\mathrm{Span}(\mathcal{T})$ and consider the matrix of $(A(x_0)B(x_0), \ldots, A(x_{2n-1})B(x_{2n-1}))$ in this base. The discriminant is the Vandermonde on $x_0, \ldots, x_{2n-2}$, hence not null. As a consequence, Algorithm 1 detects that $\mathrm{Span}(\mathcal{T})$ contains

a basis formed of vectors in $\mathcal{G}$ and stops after analysing only the root of the exploration tree. In conclusion, in the cases when Toom-Cook formulae exist, Algorithm 1 outputs them after almost no computations. A particular case of the above discussion is that of $K = GF(2)$ and $n = 2$, for which Algorithm 1 outputs the unique solution of the problem, the one presented by Karatsuba in 1962.

When it can be used, the interpolation procedure (Toom-Cook) completely solves the problem of multiplying polynomials with a minimal number of products. The next step is to minimize the number of additions needed in the algorithm. The best algorithms known carefully choose elements $x_0, \ldots, x_{2n-2}$ such that some of the computations for $A(x_0), \ldots, A(x_{2n-2})$ and $B(x_0), \ldots, B(x_{2n-2})$ are shared (see [Bod07]). For example the Fast Fourier Transform evaluates n polynomials at $2n-1$ points in time $\mathcal{O}(n \log n)$.

6 Orbits, symmetric forms and commutative formulae

In order to speed up Algorithm 1 we are tempted to use the symmetries of the problem. In this section we list the different types of symmetries. We also give a hint of the speed-up obtained when using each type of symmetry thanks to a simple approximation of the complexity: $(\dim V)\binom{\#\mathcal{G}}{k-\dim \mathcal{T}}$.

Symmetric forms. In the case of polynomial multiplication, when the degrees $n-1$ and $m-1$ of the two polynomials are equal, all the target vectors are in the subspace Sym_n of symmetric bilinear forms. Indeed, a_0b_0, $a_0b_1 + a_1b_0$, ... correspond to symmetric matrices. In this case, one can choose to replace $\mathcal{G}$ by $\mathcal{G}_{\mathrm{Sym}} := \mathcal{G} \bigcap \mathrm{Sym}_n$, i.e., products of type $(\mu_0 a_0 + \cdots + \mu_{n-1}a_{n-1})(\mu_0 b_0 + \cdots + \mu_{n-1}b_{n-1})$. The speedup is considerable since $\#\mathcal{G} = (2^n-1)^2$ and $\#\mathcal{G}_{\mathrm{Sym}} = 2^n - 1$. The price to pay is that the algorithm becomes heuristic since the larger set $\mathcal{G}$ offers *a priori* a better solution than $\mathcal{G}_{\mathrm{Sym}}$. Formally, the problem is called finding the connexion between the rank and the symmetric rank of the tensor associated to the polynomial multiplication of degree $n-1$, but this is an open problem [CGLM08].

Orbits Let now τ be an endomorphism of V such that $\tau(T) = T$ and $\tau(\mathcal{G}) = \mathcal{G}$. For example, in the case of the multiplication of an n-term by an m-term polynomial, switching coefficients of a_ib_j and $a_{n-i-1}b_{m-j-1}$ in the products can play the role of τ. Indeed, for all $u \in [0, n+m-2]$, we have $\tau(c_u) = \tau(\Sigma_{i+j=u, i\leq n-1, j\leq m-1} 1 \cdot a_ib_j) = \Sigma_{i+j=u, i\leq n-1, j\leq m-1} 1 \cdot a_{n-1-i}b_{m-1-j} = \Sigma_{i'+j'=n+m-2-u, i'\leq n-1, j'\leq m-1} a_{i'}b_{j'} = c_{n+m-2-u}$. Similarly, in the case of 2×2 matrix multiplication, switching the coefficients of $a_{i,j}b_{i',j'}$ with $a_{3-j,3-i}b_{3-j,3-i}$ for all $i, j, i', j' \in [1,2]$ sends $c_{i,j}$ in a different element

of $\mathcal{T}$ for all $i, j \in [1, 2]$. More details about the group of $\tau's$ can be read in [DG78].

One checks that, for all endomorphisms τ, for all optimal solutions $[h_0, \ldots, h_{k-1}]$, the formula $[\tau(h_0), \ldots, \tau(h_{k-1})]$ is also an optimal solution. Therefore τ acts on the one hand on the set of generators and on the other hand on the set of solutions. Since the set of optimal solutions can be divided into orbits, and since in many examples there is only a small number of optimal solutions, we make the heuristics that some of the optimal solutions are stabilized by one or more $\tau's$. This translates into the fact that an optimal solution $[h_0, \ldots, h_{k-1}]$ is the same as $[\tau(h_0),\ldots,\tau(h_{k-1})]$, or equivalently, if $h \in \{h_0, \ldots, h_{k-1}\}$, then all the orbit of h is included in $\{h_0, \ldots, h_{k-1}\}$. In the case of the naive algorithm, this translates into adding all the orbit, when one generator is added. In algorithm 1 we do the same, but we do not guarantee to find all the solutions.

Another acceleration of algorithm 1 using orbits is to reduce the number of nodes in the tree at the first level. Indeed, if we explore the formulae with a certain h_0, it is not worth to explore the formulae of type $[\tau(h_0), \ldots]$ for some τ stabilizing $\mathcal{G}$ and $\mathcal{T}$.

In terms of time, adding elements in the generated space orbit by orbit replaces $k - \dim\mathcal{T}$ by roughly $\frac{k-\dim\mathcal{T}}{\#orbit}$ in the complexity formula.

Commutative formulae. We say that a formula is non-commutative if it works for all rings R and for all assignments to $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{m-1}$ of values in R. For example, the Strassen formula is correct if one replaces variables a_i and b_j by blocks of matrices (which do not commute). It is easy to see that commutativity makes no difference in the case of bilinear computations: the elements of both $\mathcal{G}$ and $\mathcal{T}$ can be written as $\Sigma_{z=1}^{r} \lambda_z a_{i_z} b_{j_z}$, so the products $b_j a_i$ are not used. If one generalizes the problem and allows $\mathcal{G}$ to contain quadratic forms, we ignore the consequences of commutativity on the number of optimal solutions. For example, the formula $4ab = (a + b)^2 - (a - b)^2$ increases the number of solutions.

A further generalization is to search for formulae using quadratic forms in order to compute a target set of quadratic forms, like the polynomial squares. In this case, the optimal number of products to use depends on the type of formulae searched: commutative or not. Indeed, let us consider the computation of the square of a 4×4 matrix A by cutting it in four 2×2 blocks. In this case we have four target quadratic forms: $C_{1,1} = A_{1,1}^2 + A_{1,2}A_{2,1}$, $C_{1,2} = (A_{1,1} + A_{2,2})A_{1,2}$, $C_{2,1} = A_{2,1}(A_{1,1} + A_{2,2})$ and $C_{2,2} = A_{1,2}A_{2,1} + A_{2,2}^2$. The subspace $W = \text{Span}\{C_{1,2}, C_{2,1}, A_{1,1}^2, A_{2,2}^2, A_{2,1}A_{1,2}\}$ is not a non-commutative solution. Indeed, we would need $A_{2,1}A_{1,2} = A_{1,2}A_{2,1}$ for all possible values of blocks $A_{1,2}$ and $A_{2,1}$. Nevertheless, W is a commutative solution, since it can be used for the square of a matrix $A \in \mathcal{M}_2(\text{GF}(3))$ whose blocks commute.

The above discussion shows how to modify Algorithm 1 when we search commutative formulae for squaring: every time when the algorithm adds a quadratic form $g = (\mu_0 a_0 + \ldots \mu_{n-1} a_{n-1})(\nu_0 a_0 + \ldots + \nu_{n-1} a_{n-1})$ to a sub-space W, it also adds for free $g' = (\nu_0 a_0 + \ldots + \nu_{n-1 a_{n-1}})(\mu_0 a_0 + \ldots + \mu_{n-1} a_{n-1})$. The algorithm is faster when searching for squaring formulae than for products because in general squaring formulae require less generators. Moreover, when implementing the algorithm, one might represent products with only the coefficients of $a_i a_j$ with $i \leq j$. The speed-up corresponds to replacing $\dim V$ by roughly $\frac{\dim V}{2}$.

7 Alternative methods for finding upper bounds

We saw in former sections that Algorithm 1 finds *the* optimal formulae for bilinear and square forms. Nevertheless, in cases when the computations are too long, it is interesting to find formulae which are not optimal, but close to it. For example, Montgomery's 5-term formula uses 13 products compared to 14 obtained in [FH07], but the latter formula was found by a different technique, called Chinese Remainder Theorem Method [Win80], which is faster. The method has a strong connection with a generalization of Toom-Cook to extension fields [CKO09]. The CRT method was further improved using a simple relation: $A(x)B(x) \equiv c_0 + c_1 f(x) + \ldots + c_{\ell-1} f(x)^{\ell-1} \mod f(x)^\ell$ (see [CO09]). Indeed, this allows us to compute $c_0, \ldots, c_\ell$ instead of all the coefficients of $A(x)B(x) = C(x) = \Sigma_{i=0}^{2\ell-1} c_i x^i$. See Section 8 for more details.

Another trick, which is different in nature from CRT, is to find formulae for some of the coefficients of $C(x) = A(x)B(x)$ and to use Toom-Cook for the others [CKO09]. We call partial formula any formula which computes a subset of $\mathcal{T}$. Once a partial formula is found, we add the useful products to $\mathcal{T}$ and run Algorithm 1 on the new target set. See Section 8 for more details.

The exhaustive search and the above methods can be combined to obtain new formulae. For example, when Montgomery changed the upper bound for multiplication of 5-term polynomials, it had consequences on other multiplications of small degree.

8 Other applications

Matrix multiplication, middle product, cross product, squarings, ... See also Exercise 1.18 from [BZ10].

Partial formulae. In [CKO09] the authors propose a partial formula for the coefficients c_0, c_1 and c_2 of the product $C(x) = A(x)B(x)$ of two polynomials of degree above 2 (see Proposition 3 in the same article). We can

find the trick with Algorithm 1 by taking $n = m = 2$, $K = \mathrm{GF}(2)$ and $\mathcal{T} = \{a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0\}$. We find no solution with 4 products, but we find several with 5. For example $g_0 = a_0b_0$, $g_1 = a_1b_1$, $g_2 = a_2b_2$, $g_3 = (a_0 + a_1)(b_0 + b_1)$ and $g_4 = (a_0 + a_2)(b_0 + b_2)$.

Lambda function. In [CO09] the authors define the lambda function as it follows. For all prime power q, for all integer ℓ, call $\lambda_q(\ell)$ the minimal number of products needed in order to compute c_0, ..., $c_{\ell-1}$ where $\Sigma_{i=0}^{2\ell-1} c_i x^i = A(x)B(x)$. We run Algorithm 1 for $K = \mathrm{GF}(q)$ and $\mathcal{G} = \{c_0, \ldots, c_{\ell-1}\}$. We find $\lambda_2(2) = 3$ and $\lambda_2(3) = 5$ which transforms into equality the inequality in Proposition 3.4 of the same article.

Cross product.

Theorem 2. *In any field K, the minimal number of products in a bilinear computation of the cross product is* 5.

Proof. By running Algorithm 1 on the field $\mathbb{Q}$ for the set $\mathcal{G}$ of products with integer coefficients in $\{-1, 0, 1\}$ we find a solution with 5 products (there are 7072 different formulae). Therefore 5 is an upper bound. Let us show that 5 is a lower bound by a method of Michel Quercia [Que09]. Let us call ϕ the cross product. Suppose that there exist 8 linear forms on K^3 $f_0, f_1, f_2, f_3, g_0, g_1, g_2, g_3$ such that $f_0(a_0, a_1, a_2) \cdot g_0(b_0, b_1, b_2), f_1(a_0, a_1, a_2) \cdot g_1(b_0, b_1, b_2), f_2(a_0, a_1, a_2) \cdot g_2(b_0, b_1, b_2), f_3(a_0, a_1, a_2) \cdot g_3(b_0, b_1, b_2)$ span the three coordinates of the cross product i.e., $c_0 = a_1b_2 - a_2b_1$, $c_1 = a_2b_0 - a_0b_2$ and $c_2 = a_0b_1 - a_1b_0$. Let $(\lambda_{s,l})_{s\in[0,2],l\in[0,3}$ be scalars such that for all s and l, $c_s = \Sigma_{l=0}^{3} \lambda_{s,l} f_l(a_0, a_1, a_2) g_l(b_0, b_1, b_2)$. Since the cross product is not null, one of the f_i, say f_0, is not null. If g_1, g_2 and g_3 were all proportionate to f_0 then, for all $(a_0, a_1, a_2), (b_0, b_1, b_2) \in \ker f_0$ the cross product of (a_0, a_1, a_2) and (b_0, b_1, b_2) would be null. A short exercise shows that this cannot be the case (the cross product has no isotropic plane). Therefore, one of the g_1, g_2, g_3, say g_1, is not proportional to f_0. For all $(a_0, a_1, a_2) \in \ker f_0$ and $(b_0, b_1, b_2) \in \ker g_1$ we have:

$$\begin{aligned} c_0(\underline{a}, \underline{b}) &= \lambda_{0,2} f_2(\underline{a}) g_2(\underline{b}) + \lambda_{0,3} f_3(\underline{a}) g_3(\underline{b}) \\ c_1(\underline{a}, \underline{b}) &= \lambda_{1,2} f_2(\underline{a}) g_2(\underline{b}) + \lambda_{1,3} f_3(\underline{a}) g_3(\underline{b}) \\ c_2(\underline{a}, \underline{b}) &= \lambda_{2,2} f_2(\underline{a}) g_2(\underline{b}) + \lambda_{2,3} f_3(\underline{a}) g_3(\underline{b}) \end{aligned}$$

where $\underline{a}$ and $\underline{b}$ stay for (a_0, a_1, a_2) and (b_0, b_1, b_2) respectively. Therefore, $\phi(\ker f_0 \times \ker g_1)$ is included in $\mathrm{Span}\langle(\lambda_{0,2}, \lambda_{1,2}, \lambda_{2,2}), (\lambda_{0,3}, \lambda_{1,3}, \lambda_{2,3})\rangle$, which is of dimension at most 2. On the other hand, it is an short exercise to show for all distinct planes P and Q that $\mathrm{Span}(\phi(P \times Q)) = K^3$. Hence, we obtain a contradiction, so no formula with 4 products exists. □

Multiplication by squaring. Thanks to formula $ab = \frac{1}{4}[(a+b)^2-(a-b)^2]$ one can obtain an algorithm which multiplies polynomials in $Z[X]$ with no division other than by 4 and of complexity $\mathcal{O}(n^{\log_2 3})$ [Kar95]. Indeed, the algorithm uses recursively the formula $2\alpha\beta = (\alpha+\beta)^2 - \alpha^2 - \beta^2$ to compute the square of an 2^s-term polynomial in $\mathcal{O}(3^s)$ operations. A generalization of the method is to compute a polynomial square by making only squares. Let us call $k(n)$ the minimal number of squares necessary over GF(3) to compute the square of an n-term polynomial. Thanks to Algorithm 1 we obtained the following results:

n	2	3	4	5
$k(n)$	3	6	9	12

The last column gives a multiplication algorithm of complexity $\mathcal{O}(n^{\log_5 12})$, asymptotically faster than the ones obtained using the multiplication formulae without squares. Nevertheless, we do not expect to see the advantage in the size range where Karatsuba-Ofman's algorithm beats FFT. Moreover, the multiplication by squaring has large hidden constants and provides commutative formulae (see Section 6).

Computing tensor rank. Algorithm 1 computes the bilinear (quadratic) complexity of a set of bilinear (quadratic) forms by reducing it to the problem of evaluating the rank of a specific order-3 tensor. Therefore, Algorithm 1 can be used for evaluating the rank of any order-3 tensor. For example a test made on a random sample of 100 tensors 2×2 over GF(2) showed that 1% have rank 0, 8% rank 1, 65% rank 2, 26% rank 3 and none has rank 4 or more. The same experiment over GF(3) revealed that 0% have rank 0, 4% rank 1, 65% rank 2, 31% rank 3 and none rank 4 or more.

9 Experimental results

We will try here to improve the results of [CÖ10], and to reproduce the formulae from [CH07] for squarings.

Table 1 gives experimental results over GF(2), for example for 6×6 the first row "None" says that with no constraint, there is no formula with only $k = 14$ products over GF(2). The second row "Orb" says that when we include both $a_i b_j$ and $a_{5-i} b_{5-j}$ in the products, then with $k = 17$ products there are 2 formulae. The third row "Sym" says that when we use only symmetric bilinear forms, such as $(a_5+a_0)(b_5+b_0)$, and disallow for example $(a_5 + a_0)(b_4 + b_1)$, then with $k = 17$ there are 6 formulae.

Table 2 gives similar results over GF(3). The last row 8×8 says that with only symmetric bilinear forms, including both $a_i b_j$ and $a_{7-i} b_{7-j}$ in the products, there is no formula with $k = 22$ products.

Finally Table 3 corresponds to small extensions of GF(2) or GF(3), where we consider the multiplication of two elements in polynomial basis.

10 Conclusion

When looking back to how the known "magical" formulae were discovered, one realizes that many of them are a combination of advanced mathematical theories and creativity. As an isolated exception, the technique in [Mon05], that we improve here, uses simple notions and does not interpret its formulae. In order to compare the two techniques, let us consider the results in Section 9. When multiplying two 3-term polynomials there are 9 optimal formulae, out of which 7 are the Toom-Cook formula or can be derived easily from it. The last two though are surprising because they were not published before the extensive search found them. Moreover, from the 4 formulae associated to 4-term polynomials only one was known: the recursive use of Karatsuba's formula. Finally, the best formula obtained by the mathematical method of Chinese Remainder Theorem for 5-term polynomials uses 14 products ([FH07]) even though there are at least 21 formulae with 13 products. Nevertheless, searching for formulae is closely related to computing the rank of a tensor [Str88], which was proven to be NP-hard [Has90]. Therefore, apart from improving the formulae for squaring a 4-term polynomial, inventing new formulae for 8-term polynomial multiplication and proving that the formulae for 2, 3 and 4 polynomial multiplication, small sizes squaring, cross product, quaternion product and other formulae are optimal, the extensive search cannot provide new results. If one wants to go further, one has to speed up the computations by either restricting the set $\mathcal{G}$ of products, e.g., the symmetric products for polynomial multiplication, or by enlarging the target set $\mathcal{T}$ by imposing that some products have to occur in the formula.

References

[Art91] M. Artin. *Algebra*. Prentice-Hall, Inc., 1991.

[Bod07] M. Bodrato. Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. *Arithmetic of Finite Fields*, pages 116–133, 2007.

[BZ10] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Number 18 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010. Electronic version freely available at `http://www.loria.fr/~zimmerma/mca/pub226.html`.

Table 1: Experimental results for $n \times m$ polynomial multiplication over GF(2)[X].

$n \times m$	Constraints	$\#\mathcal{G}$	k	# of leaves	# of solutions	Calculation time [s]
2×2	None	9	3	1	1	0.00
3×2	None	21	5	2	2	0.00
3×3	None	49	6	9	2	0.00
4×2	None	45	6	5	3	0.00
4×3	None	105	8	700	30	0.01
4×4	None	225	9	$6.60 \cdot 10^3$	4	0.10
5×2	None	93	8	56	27	0.00
5×3	None	217	10	$1.46 \cdot 10^5$	363	1.96
5×4	None	465	12	$3.13 \cdot 10^8$	3 798	$1.37 \cdot 10^4$
5×5	None	961	13	$9.65 \cdot 10^9$	24	$9.90 \cdot 10^5$
	Orb.	505	13	$3.34 \cdot 10^5$	3	16.1
	Sym.	31	13	$2.10 \cdot 10^3$	20	0.01
6×2	None	189	9	250	61	0.00
6×3	None	441	11	$2.05 \cdot 10^6$	2	72.6
6×4	None	945	13	$7.69 \cdot 10^9$	8	$6.83 \cdot 10^5$
6×5	Orb.	1 001	16	$3.70 \cdot 10^8$	30	$4.43 \cdot 10^4$
6×6	None	3 969	14	$4.37 \cdot 10^9$	0	$1.85 \cdot 10^6$
	Orb.	2 009	17	$2.24 \cdot 10^9$	2	$6.53 \cdot 10^5$
	Sym.	63	17	$8.08 \cdot 10^6$	6	54.3
7×2	None	381	11	$9.14 \cdot 10^3$	956	0.23
7×3	None	889	13	$2.52 \cdot 10^9$	87	$1.99 \cdot 10^5$
7×4	Orb.	975	15	$1.31 \cdot 10^7$	1	$1.39 \cdot 10^3$
7×5	Orb.	2 021	17	$8.52 \cdot 10^9$	0	$1.94 \cdot 10^6$
7×6	—	—	—	—	—	—
7×7	Sym.	127	22	$3.42 \cdot 10^{12}$	2 460	$5.43 \cdot 10^7$
	Sym. + Orb.	71	22	$8.68 \cdot 10^6$	20	97.5
8×2	None	765	12	$7.80 \cdot 10^4$	4 080	3.58
8×3	Orb.	915	15	$6.56 \cdot 10^6$	38	655
8×4	Orb.	1 935	17	$1.28 \cdot 10^9$	2	$3.28 \cdot 10^5$
8×5	—	—	—	—	—	—
8×6	—	—	—	—	—	—
8×7	—	—	—	—	—	—
8×8	Sym. + Orb.	255	26	$4.67 \cdot 10^9$	6	$1.33 \cdot 10^5$

Table 2: Experimental results for $n \times m$ polynomial multiplication over GF(3)[X].

$n \times m$	Constraints	$\#\mathcal{G}$	k	# of leaves	# of solutions	Calculation time [s]
2×2	None	16	3	1	1	0.00
3×2	None	52	4	1	1	0.00
3×3	None	169	6	24	13	0.00
4×2	None	160	6	9	9	0.00
4×3	None	520	7	164	7	0.01
4×4	None	1 600	9	$4.11 \cdot 10^5$	595	61.9
5×2	None	484	7	24	21	0.00
5×3	None	1 573	9	$2.80 \cdot 10^5$	1 041	42.5
5×4	None	4 840	10	$4.75 \cdot 10^6$	44	$3.25 \cdot 10^3$
5×5	None	14 641	11	$4.89 \cdot 10^7$	0	$1.09 \cdot 10^5$
	Orb.	7 465	12	$3.75 \cdot 10^6$	30	$4.34 \cdot 10^3$
	Sym.	121	12	$3.93 \cdot 10^4$	31	0.71
6×2	None	1 456	8	69	44	0.01
6×3	None	4 732	10	$3.24 \cdot 10^6$	227	$2.04 \cdot 10^3$
6×4	Orb.	7 384	12	$1.96 \cdot 10^6$	27	$1.92 \cdot 10^3$
6×5	—	—	—	—	—	—
6×6	Sym.	364	15	$2.37 \cdot 10^8$	3	$1.72 \cdot 10^4$
7×2	None	4 372	10	$2.27 \cdot 10^4$	10 051	9.80
7×3	Orb.	7 237	12	$1.82 \cdot 10^6$	236	$1.70 \cdot 10^3$
7×4	—	—	—	—	—	—
7×5	—	—	—	—	—	—
7×6	—	—	—	—	—	—
7×7	Sym.	1 093	16	$1.03 \cdot 10^8$	0	$2.15 \cdot 10^4$
	Sym. + Orb.	573	19	$2.28 \cdot 10^8$	4	$3.19 \cdot 10^4$
8×2	None	13 120	11	$2.01 \cdot 10^5$	80 954	355
8×3	—	—	—	—	—	—
8×4	—	—	—	—	—	—
8×5	—	—	—	—	—	—
8×6	—	—	—	—	—	—
8×7	—	—	—	—	—	—
8×8	Sym. + Orb.	1 680	22	$1.54 \cdot 10^{11}$	0	$5.66 \cdot 10^7$

Table 3: Experimental results for multiplication over small extensions of GF(2) or GF(3).

Finite field	**Constraints**	$\#\mathcal{G}$	k	**# of leaves**	**# of solutions**	**Calculation time [s]**
$GF(2^2)$	None	9	3	3	3	0.00
$GF(2^3)$	None	49	6	$7.03 \cdot 10^3$	105	0.02
$GF(2^4)$	None	225	9	$2.57 \cdot 10^9$	2 025	955
$GF(2^5)$	None	961	9	$3.10 \cdot 10^{10}$	0	$1.83 \cdot 10^6$
	Sym.	31	13	$3.49 \cdot 10^6$	2 015	13.7
$GF(2^6)$	Sym.	63	14	$3.78 \cdot 10^9$	0	$2.50 \cdot 10^5$
$GF(2^7)$	Sym.	127	14	$8.93 \cdot 10^{10}$	0	$1.22 \cdot 10^6$
$GF(3^2)$	None	16	3	4	4	0.00
$GF(3^3)$	None	169	6	$2.42 \cdot 10^5$	11 843	5.35
$GF(3^4)$	None	1 600	7	$6.29 \cdot 10^8$	0	$1.16 \cdot 10^5$
	Sym.	40	9	$1.10 \cdot 10^5$	234	0.98
$GF(3^5)$	Sym.	121	10	$1.83 \cdot 10^8$	0	$3.77 \cdot 10^3$

[CA66] S.A. Cook and S.O. Aanderaa. *On the minimum computation time of functions*. Harvard University, 1966.

[CGLM08] P. Comon, G. Golub, L. Lim, and B. Mourrain. to appear in siam journal on matrix analysis and applications. symmetric tensors and symmetric tensor rank. 2008.

[CH07] J. Chung and M. A. Hasan. Asymmetric squaring formulae. In P. Kornerup and J.-M. Muller, editors, *Proc. of the 18th IEEE Symposium on Computer Arithmetic (ARITH'18)*, pages 113–122. IEEE Computer Society, 2007.

[CKO09] M. Cenk, C.K. Koç, and F. Ozbudak. Polynomial multiplication over finite fields using field extensions and interpolation. In *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pages 84–91. IEEE, 2009.

[CO09] M. Cenk and F. Ozbudak. Improved polynomial multiplication formulas over GF(2) using chinese remainder theorem. *Computers, IEEE Transactions on*, 58(4):572–576, 2009.

[CÖ10] Murat Cenk and Ferruh Özbudak. On multiplication in finite fields. *J. Complexity*, 26:172–186, 2010.

[DG78] H.F. De Groote. On varieties of optimal algorithms for the computation of bilinear mappings i. the isotropy group of a bilinear mapping. *Theoretical Computer Science*, 7(1):1–24, 1978.

[FH07] H. Fan and A. Hasan. Comments on five, six, and seven-term karatsuba-like formulae'. *Computers, IEEE Transactions on*, 56(5):716–717, 2007.

[Has90] J. Hastad. Tensor rank is np-complete. *Journal of Algorithms*, 11(4):644–654, 1990.

[HQZ04] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. The middle product algorithm, I. Speeding up the division and square root of power series. *AAECC*, 14(6):415–438, 2004.

[Kar95] AA Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics*, 211:169–183, 1995.

[Knu81] D.E. Knuth. *The Art of Computer Programming (Volume 2)*. Addison–Wesley, 1981.

[KO62] Anatolii A. Karatsuba and Yuri Ofman. Multiplication of multi-digit numbers on automata (in Russian). *Doklady Akad. Nauk SSSR*, 145(2):293–294, 1962. Translation in *Soviet Physics-Doklady* 7 (1963), 595–596.

[Mon05] P.L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, pages 362–369, 2005.

[Que09] Michel Quercia. personal communication to Paul Zimmermann, 2009.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[Str88] V. Strassen. The asymptotic spectrum of tensors. *Journal f "ur die reine und angewandte Mathematik (Crelles Journal)*, 1988(384):102–152, 1988.

[Too63] A.L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[Win80] S. Winograd. *Arithmetic complexity of computations*. Number 33. Society for Industrial Mathematics, 1980.